Performing Infrastructure Migrations at Airbnb Scale
Who am I?
AIRBNB CASE STUDY
Kubernetes (k8s) is an open-source system for automating deployment, scaling, and management of containerized applications.
70% of services in kubernetes
300+ critical services in kubernetes
WHAT ARE MIGRATIONS
Example Migrations

- non-cloud to cloud
- VMs to containers
- configuration management to orchestration
- API framework changes (ex: circuit breaking, request throttling)
- new CI, build, or deploy system
- new service proxy or service mesh
- new language/framework version
- security patches
- ... and more!
Migrations

"LOW" TO "HIGH" EFFORT

low effort
- API framework change
- security patch
- deprecate endpoint

high effort
- OS upgrade
- new CD system
- non-cloud to cloud
- VMs to containers
- orchestration (k8s)
- language upgrade
- new storage system
- new service mesh
- new CI system
- language upgrade
- new storage system
- new service mesh
- deprecate endpoint
Migrations

“URGENT” AND “DISCRETIONARY”

- OS upgrade
- security patch
- API framework change
- upgrade JVM version
- new service mesh
- new storage system
- new CI system
- new CD system
- deprecate endpoint
- language upgrade
- VMs to containers
- non-cloud to cloud
- orchestration (k8s)
Migrations

NEED AT "LOW" VS "HIGH" SCALE

<table>
<thead>
<tr>
<th>Low Scale</th>
<th>High Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>language upgrade</td>
<td>upgrade JVM version</td>
</tr>
<tr>
<td>security patch</td>
<td>non-cloud to cloud</td>
</tr>
<tr>
<td>OS upgrade</td>
<td>orchestration (k8s)</td>
</tr>
<tr>
<td></td>
<td>language upgrade</td>
</tr>
<tr>
<td></td>
<td>upgrade JVM version</td>
</tr>
<tr>
<td></td>
<td>non-cloud to cloud</td>
</tr>
<tr>
<td></td>
<td>orchestration (k8s)</td>
</tr>
<tr>
<td></td>
<td>new CI system</td>
</tr>
<tr>
<td></td>
<td>API framework change</td>
</tr>
<tr>
<td></td>
<td>new service mesh</td>
</tr>
<tr>
<td></td>
<td>new storage system</td>
</tr>
</tbody>
</table>
Migrations are multidimensional.
Migrations are multidimensional.
WHY ARE MIGRATIONS IMPORTANT
Migrations reduce tech debt
Examples of tech debt

- low developer velocity
- slower CI, builds, deploys
- non-reproducibility
- reduced resiliency
- hitting scaling limits
- networking issues
- security holes
- outdated language/framework version
- end-of-life systems
- ... and more!
Migrations are the sole lever to systematically create technical leverage at scale
MIGRATION STRATEGIES
MIGRATION TYPES
Migration types

- **component**: upgrades, patches, refactors
- **system**: move from one system to another
- **infrastructure**: rewrite underlying infra
Migration types
AIRBNB EXAMPLE

- **component:** security patch, upgrade ruby, upgrade ubuntu, deprecate endpoint
- **system:** new CI/CD system, new build system, deployment pipelines, new proxy, new load balancer, new service mesh, new storage
- **infrastructure:** move to cloud, containerization, k8s, change cloud provider
Migration types

- **know which type** you’re dealing with
- exponentially **increasing complexity** for each type
- more complex means **more resourcing required** and **more risk**
MIGRATION SEQUENCING
Sequenced Migrations

- complex migrations likely have dependent migrations
- requires planning: migration sequencing
- or infrequent cascading migrations and inefficient simultaneous migrations
Cascading Migrations

- **Infrequent migrations** can cause a rewrite or **cascading migrations**.
- Because small incremental changes are not made, a **rewrite is required** to get to the ideal end state.
- Cascading complexity means **higher risk** overall.
Cascading Migrations

**EXAMPLE**

- going from non-cloud to k8s
- for high availability, consider completing incremental migrations sequentially
Simultaneous Migrations

- Inefficient migrations can cause overall migration velocity to slow down and lead to simultaneous migrations.
- Simultaneous migrations don’t necessarily depend on each other.
- But they do affect the overall complexity of the system and introduce additional risk.
Simultaneous Migrations

EXAMPLE

- are these migrations really independent?
- could each migration be making assumptions about your system?
- does your migration need to support a mixed state from another migration?
Sequenced Migrations

**STRATEGIES**

- a minor migration now can become an infrastructure rewrite later
- make migration frequent and efficient
- tightly scoped migrations are easier
- sequenced migrations are safer
Sequenced Migrations

- lower risk migrations
- requires more planning and time
- can parallelize migrations without dependencies
Sequenced Migrations

EXAMPLE

- start with JVM upgrade and service discovery layer in parallel
- then complete containerization effort
Prioritized Migrations

**STRATEGIES**

- **prioritize** migrations that reduce or simplify further migrations
- example: migrate to “infrastructure as code” first
- following migrations are now code refactors
MIGRATION AT SCALE
that new service mesh you were thinking about

IS SUDDENLY VERY URGENT
that new service mesh you were thinking about is suddenly very urgent

you are here
Migration at scale

• scale can introduce *increased load or traffic*
Migration at scale

- when not planned for, this can introduce **forced urgency**
Migration at scale

- operating at scale can mean you need to migrate more services, databases, etc
- increased effort comes from number of surface to migrate and complexity
Migration at scale

**EXAMPLE**

- you want to switch from using HAProxy to Envoy Proxy
- you have **exponentially more services** and edges
- issues with HAProxy **compound with more edges**
Migration at scale

STRATEGIES

- **forecast** expected load
- **stress test** systems for actual load
- get ahead of the problem with **long-term planning** before it becomes firefighting
Migration at scale

STRATEGIES

• make time work for you
• deprecate the old thing first
• make the new approach the default
Airbnb example:
- moving monolithic service configuration to their service codebases
- **exponentially more services** are being created
- create a service generator that generates services using new approach
MIGRATION OVERHEAD
Migration overhead

- Migration start
- Unfinished migration
- Migration end

Overhead

Time
migrating is an explicit tradeoff of taking on overhead now to reduce worse overhead later.
Migration overhead

- for those **running** the migration effort
- for those **migrating** to the “new” thing
- for those **maintaining** both the “new” and “old” thing
Migration overhead

**EXAMPLE**

- you want to switch from using HAProxy to Envoy Proxy
- you have *exponentially more services* and edges
- you have more complexity with *different use cases* (ex: HTTP, TCP)
- you’re patching HAProxy while building out Envoy Proxy *(maintaining mixed state)*
Migration overhead: what developers get

make progress on their thing

working on all 40 “small” migration asks

90%

10%
Unfinished migrations

- migration is not 100% finished
- working on all "small" migration asks
- worsening tech debt
- start a new migration

@MELANIECEBULA
Unfinished migrations

worsening tech debt

migration is not 100% finished

start a new migration

@MELANIECEBULA
Unfinished migrations

WORST BINGO EVER

- future migrations are now harder
- tech debt is now worse instead of better
- bugs, regressions, edge cases (BINGO)

your infrastructure state diagram
unfinished migrations make tech debt worse.
Migration overhead

STRATEGIES

• develop abstractions over the infrastructure you migrate
• make the current migration easier
• avoid leaky abstractions
• makes future migrations easier
kubernetes config files

Production Deployment
Canary Deployment
Dev Deployment

Production ConfigMap
Canary ConfigMap
Dev ConfigMap

Production Service
Canary Service
Dev Service

kubectl apply

kubernetes cluster

@MELANIECEBULA

abstraction
AIRBNB EXAMPLE
generating k8s configs

kube-gen generate

kube-ctl apply

kubernetes config files

Production Deployment
Canary Deployment
Dev Deployment
Production ConfigMap
Canary ConfigMap
Dev ConfigMap
Production Service
Canary Service
Dev Service

kubernetes cluster

@MELANIECEBULA
**generating k8s abstraction**

**kube-gen generate**

**kubectl apply**

**kubernetes config files**

- Production Deployment
- Canary Deployment
- Dev Deployment
- Production ConfigMap
- Canary ConfigMap
- Dev ConfigMap
- Production Service
- Canary Service
- Dev Service

**kubernetes cluster**
better abstraction?

- service-manifest.yml
- chef config files
- kubernetes config files
- the next thing

time
better abstraction?

service-manifest.yml

smartstack

envoy proxy

the next thing

time
Migration overhead

STRATEGIES

• **standardize on the 90% use case**

• **automatically migrate** for the standard use case

• migrate under an **abstraction layer**

• **migrate programmatically** as a code refactor
How do we migrate programmatically?

- Service configuration lives alongside application code.
- Many simple migrations are automated refactors.
- Refactor process is a collection of modular scripts that cover refactor lifecycle.

```
| get-repos.py | close.py |
| list-pr-urls.py | refactor.py |
| update-prs.py | status.py |
```

refactorator
The lifecycle of a refactor

Run Refactor
Checks out repo, finds project, runs refactor job, tags owners, creates PR

Update
Comments on the PR, reminding owners to verify, edit, and merge the PR

Merge
Merges the PR with different levels of force
What do we migrate programmatically?

**EXAMPLES**

- configuration upgrades (ex: k8s version)
- base image upgrades
- security patches
- changing CI/CD system
- deprecating configuration feature
- migrating monolithic configuration to service code
MIGRATION PROGRAM
Migration strategy: make one person do it

Make one person
do all of it

one engineer enables
and completes entire
migration
Migration strategy: make one person do it

pros:

- very **tight feedback loop** for gaps in the migration process
- **easy to track and finish**

cons:

- scale makes this an impossible long-term strategy
Migration strategy: make devs do all of it

- Make devs do all of it
- devs are given timelines and asked to self-serve migration before deadline
Migration strategy: make devs do all of it

**Pros:**

- very **low overhead** for overwhelmed infra team
- distributed effort

**Cons:**

- no feedback loop for unexpected migration blockers / risks
- migrations left unfinished
Migration strategy: an actual migration program

Create a migration program

migration team owns migration end-to-end and partners with leadership and devs to finish
Migration strategy: an actual migration program

**pros:**

- migration can be systematically **enabled and vetted**
- migration can be **sequenced** with others
- **tight feedback loop** for gaps in the migration process
- distributed effort
- easy to track and **finish**
MIGRATION LIFECYCLE: VALIDATE PHASE
Validate Phase

DOES IT WORK?

- a design document
- a prototype
- tie in with overall roadmap
- stress test with early users
- *iterate until...* you’re convinced you’ve fully validated the technology
Validate Phase
AIRBNB EXAMPLE

- k8s design document
- a prototype (cluster, simple k8s service)
- tie in with service discovery migration plan
- stress test with high-latency low-throughput services
MIGRATION LIFECYCLE: ENABLE PHASE
Enable Phase

MAKE THE MIGRATION WORK

- build tooling
- build abstraction layer
- make the new thing the default
- write documentation & code labs
- programatically migrate the 90%
- **iterate until**... you’re convinced you’ve fully enabled the migration
• new project tool, CLI tool, integration with CI/CD tooling
• k8s abstraction layer
• new services are created with new project tool
• docs, code labs, and training classes
• migration tooling
MIGRATION LIFECYCLE: FINISH PHASE
Finish Phase

IS EVERYTHING CUT OVER?

- migration plan and sequencing
- programmatically migrate services
- engage with leadership
- set and track across migration goals
- work with devs to identify ongoing risks & blockers
- be prepared for migrations to get harder to finish towards the end
- *iterate until*... you’ve fully migrated to the new system
Finish Phase
AIRBNB EXAMPLE

- phased migrations to k8s
- engage with leadership across business units (dev teams)
- set and track across migration company goals
- TPM & PM work with devs to identify ongoing risks & blockers
- migration-specific documentation & tooling (80% of dev services)
- eng runs office hours to help with tricky migrations
- still working on this phase!
10 Takeaways

1. identify migration type to determine overall complexity and risk
2. run frequent, efficient, and tightly-scoped migrations
3. sequence, prioritize, and parallelize migrations
4. long-term planning, forecasting, and stress-testing to avoid surprise migrations
5. make the new approach the default
6. fully finish migrations to reduce tech debt
7. develop abstractions over infrastructure
8. run migrations as code refactors
9. run a migration program with a migration lifecycle
10. iterate on your migration to ensure its fully validated, enabled, and finished